Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 8: this Friday @ 11:59 pm
  - Assignment 2: this Friday @ 11:59 pm
    - Support video and slides on Canvas
  - Lab 7: Tuesday 3/21 @ 11:59 pm

Previous lecture

- LZW example and corner case
- Shannon’s Entropy
- LZW vs. Huffman
- Burrows-Wheeler Compression Algorithm

This Lecture

- Burrows-Wheeler Compression Algorithm
- ADT Priority Queue (PQ)
  - Heap implementation
  - Heap Sort
  - Indexable PQ
- ADT Graph
  - definitions
  - representations

Burrows-Wheeler Data Compression Algorithm

- **Best** compression algorithm (in terms of compression ratio) **for text**
- The basis for UNIX's **bzip2** tool

Adapted from: https://www.cs.princeton.edu/courses/archive/spr03/cos226/assignments/burrows.html

BWT: Compression Algorithm

- Three steps
  - Burrows-Wheeler Transform
    - Cluster same letters as close to each other as possible
  - Move-To-Front Encoding
    - Convert output of previous step into an integer file with **large frequency** differences
  - Huffman Compression
    - Compress the file of integers using Huffman Compression

BWT: Expansion Algorithm

- Apply the inverse of compression steps in reverse order
  - Huffman decoding
  - Move-To-Front decoding
  - Inverse Burrows-Wheeler Transform

Move-To-Front Encoding

- Initialize an ordered list of the 256 ASCII characters
  - character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list (i.e., index 0)

Move-To-Front Encoding

Input:

e a e d e e

0	**a**
1	**b**
2	**c**
3	**d**
4	**e**

Output:

Move-To-Front Encoding

•

Input:

e a e d e e

0	a
1	b
2	c
3	d
4	e

Output:

4

Move-To-Front Encoding

•

Input:

e a e d e e

0	**a**	**e**
1	**b**	**a**
2	**c**	**b**
3	**d**	**c**
4	**e**	**d**

Output:

4

Move-To-Front Encoding

•

Input:

e a e d e e

0	a	e	a
1	b	a	e
2	c	b	b
3	d	c	c
4	e	d	d

Output:

4 1

Move-To-Front Encoding

•

Input:

e a e d e e

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:

4 1 1

Move-To-Front Encoding

Input:

e a e d e e

0	**a**	**e**	**a**	**e**	**d**
1	**b**	**a**	**e**	**a**	**e**
2	**c**	**b**	**b**	**b**	**a**
3	**d**	**c**	**c**	**c**	**b**
4	**e**	**d**	**d**	**d**	**c**

Output:

4 1 1 4

Move-To-Front Encoding

•

Input:

e a e d e e

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

4 1 1 4 1

Move-To-Front Encoding

Input:

e a e d e e

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

4 1 1 4 1 0

Move-To-Front Encoding

In the output of MTF Encoding, smaller integers have higher frequencies than larger integers

Move-To-Front Decoding

- Initialize an ordered list of 256 characters
  - same as encoding
- For each integer *i* (*i* is between 0 and 255)
  - print the *i*th character in the list
  - move that character to the front of the list

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a
1	b
2	c
3	d
4	e

Output:

e

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e
1	b	a
2	c	b
3	d	c
4	e	d

Output:

e

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a
1	b	a	e
2	c	b	b
3	d	c	c
4	e	d	d

Output:

e a

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:

e a e

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a	e	d
1	b	a	e	a	e
2	c	b	b	b	a
3	d	c	c	c	b
4	e	d	d	d	c

Output:

e a e d

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

e a e d e

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

e a e d e e

BWT: Compression Algorithm

- **Compression**
  - Burrows-Wheeler Transform
  - Move-To-Front Encoding
  - Huffman Compression

- **Expansion**
  - Huffman decoding
  - Move-To-Front decoding
  - Inverse Burrows-Wheeler Transform

Burrows-Wheeler Transform

- **Rearranges** the characters in the input
  - lots of clusters with **repeated characters**
  - still possible to **recover** the original input
- Intuition: Consider the string **hen** in English text
  - most of the time the letter preceding it is t or w
  - group all such preceding letters together (mostly t's and some w’s)

Burrows-Wheeler Transform

- For each block of length N characters
  - generate **N strings** by **cycling** the characters of the block one step at a time
  - **sort** the strings
  - output is the **last column** in the sorted table and the **index** of the original block in the sorted array

Burrows-Wheeler Transform

- Example: Let's transform "ABRACADABRA"
- N = 11
- Cyclic Versions of the string:
  - ABRACADABRA
  - BRACADABRAA
  - RACADABRAAB
  - ACADABRAABR
  - CADABRAABRA
  - ADABRAABRAC
  - DABRAABRACA
  - ABRAABRACAD
  - BRAABRACADA
  - RAABRACADAB
  - AABRACADABR
- After Sorting
  - AABRACADABR
  - ABRAABRACAD
  - 2 → ABRACADABRA
  - ACADABRAABR
  - ADABRAABRAC
  - BRAABRACADA
  - ADABRAA
  - CADABRAABRA
  - DABRAABRACA
  - RAABRACADAB
  - RACADABRAAB

RDARCAAAABB

Burrows-Wheeler Transform Example 2

- Input: ABABABA
- **Step 1: Build an array of 7 strings, each a circular rotation of the original by one character**
- ABABABA
- BABABAA
- ABABAAB
- BABAABA
- ABAABAB
- BAABABA
- AABABAB
- **Step 2: Sort the array alphabetically**
- **Notice that** the first column of the sorted array has the same characters as the last column
  - all columns have the same set of letters
- **Step 3: Output the last column of the sorted array and the index of the input string in the sorted array**

original array	sorted array
ABABABA	AABABAB
BABABAA	ABAABAB
ABABAAB	ABABAAB
BABAABA	ABABABA
ABAABAB	BAABABA
BAABABA	BABAABA
AABA	BAA

Output of BWT:

BBBAAAA and 3

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 1: Sort the encoded string**
  - BBBAAAA → AAAABBB
  - The first column of the sorted array has the same characters as the last column
    - but in sorted order

??????	B
??????	B
??????	B
??????	A
??????	A
??????	A
??????	A

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 1: Sort the encoded string**
  - BBBAAAA → AAAABBB
  - This gives us the first column of the sorted array

A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - holds the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

original array

ABABABA
BABABAA
ABABAAB
BABAABA
ABAABAB
BAABABA
AABABAB

A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

	next
A?????B	-
A?????B	-
A?????B	-
A?????A	-
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

	next
A?????B	3
A?????B	-
A?????B	-
A?????A	-
B?????A	-
B?????A	-
B?????A	-

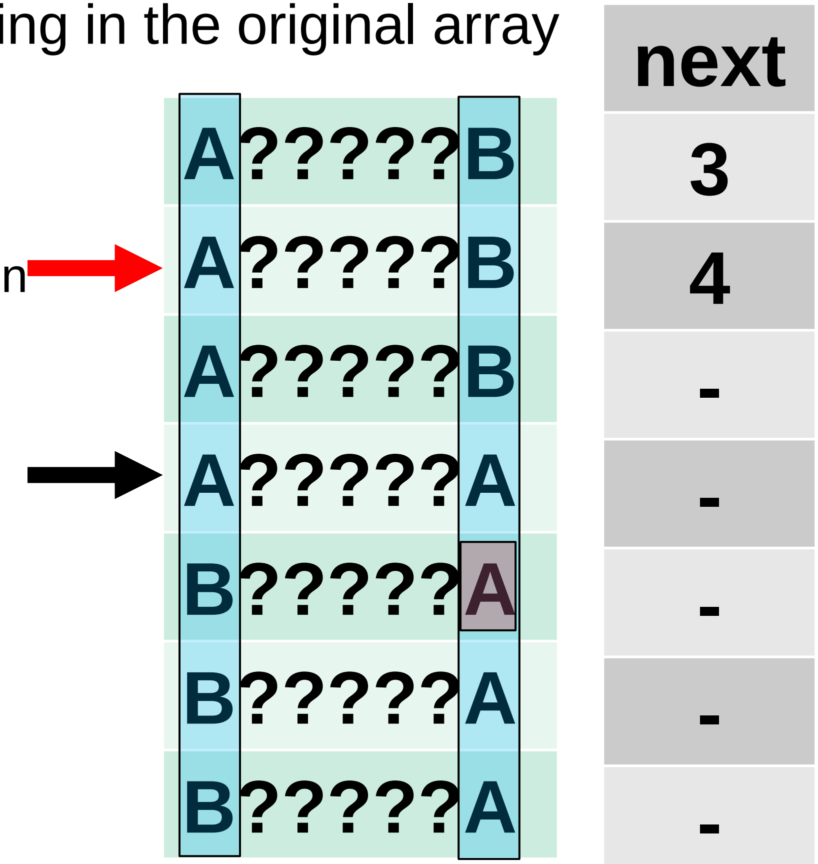
Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

	next
A?????B	3
A?????B	4
A?????B	-
A?????A	-
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

	next
A?????B	3
A?????B	4
A?????B	5
A?????A	-
B?????A	-
B?????A	-
B?????A	-

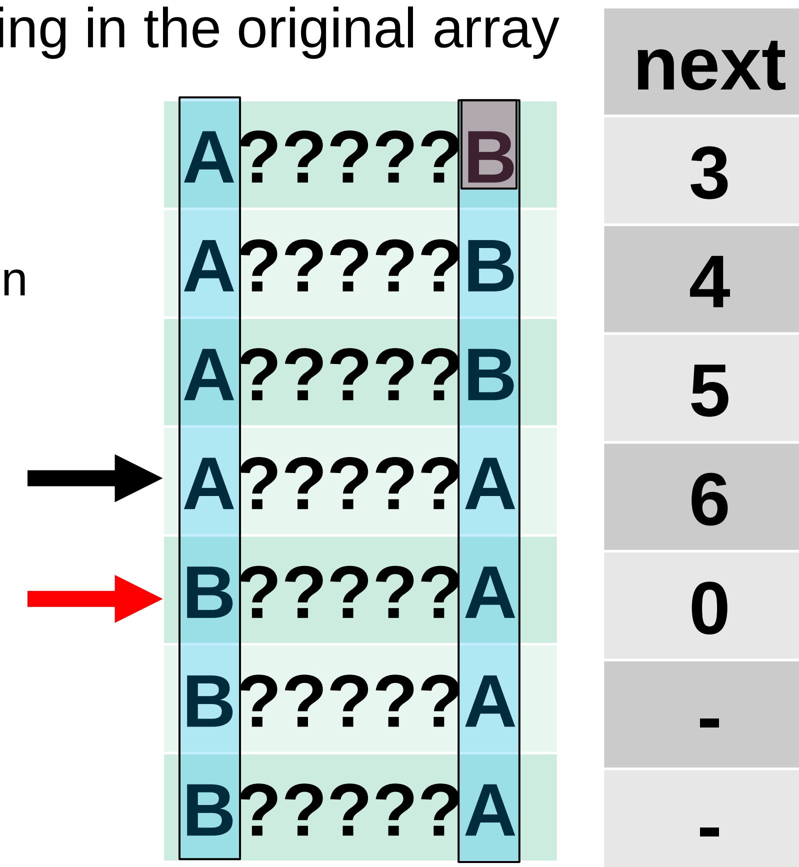
Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

			next
A	?????	B	3
A	?????	B	4
A	?????	B	5
A	?????	A	6
B	?????	A	-
B	?????	A	-
B	?????	A	-

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	-

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

			next
A	?????	B	3
A	?????	B	4
A	?????	B	5
A	?????	A	6
B	?????	A	0
B	?????	A	1
B	?????	A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column
- Why does that work?
    - first character of a string becomes the last character in the next string in the original order

	next
A?????B	3
A?????B	4
A?????B	5
→ A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?

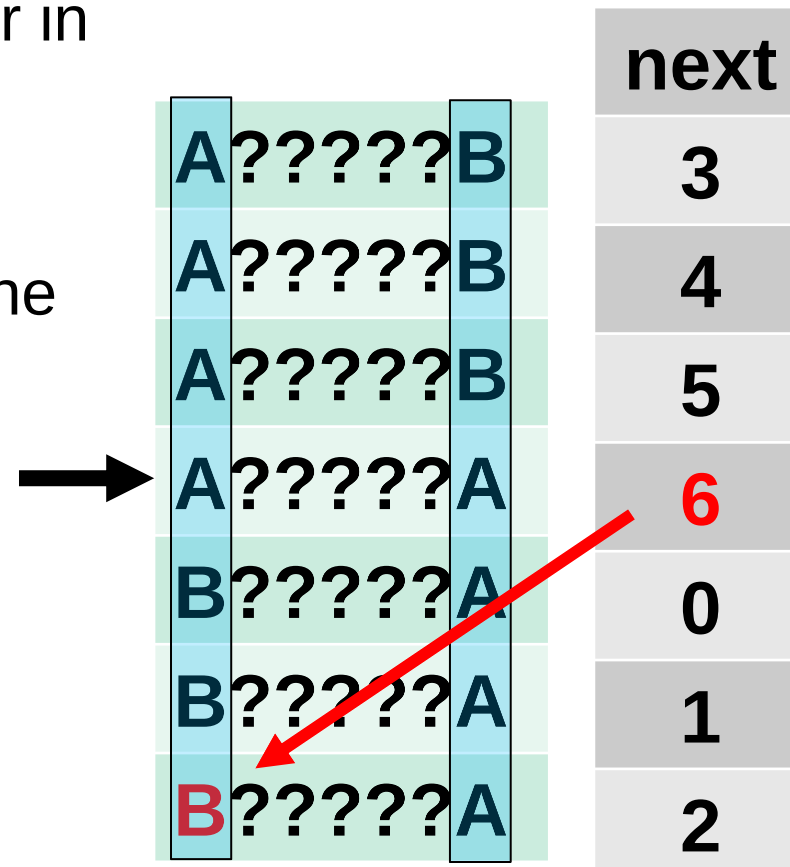
A??????

	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

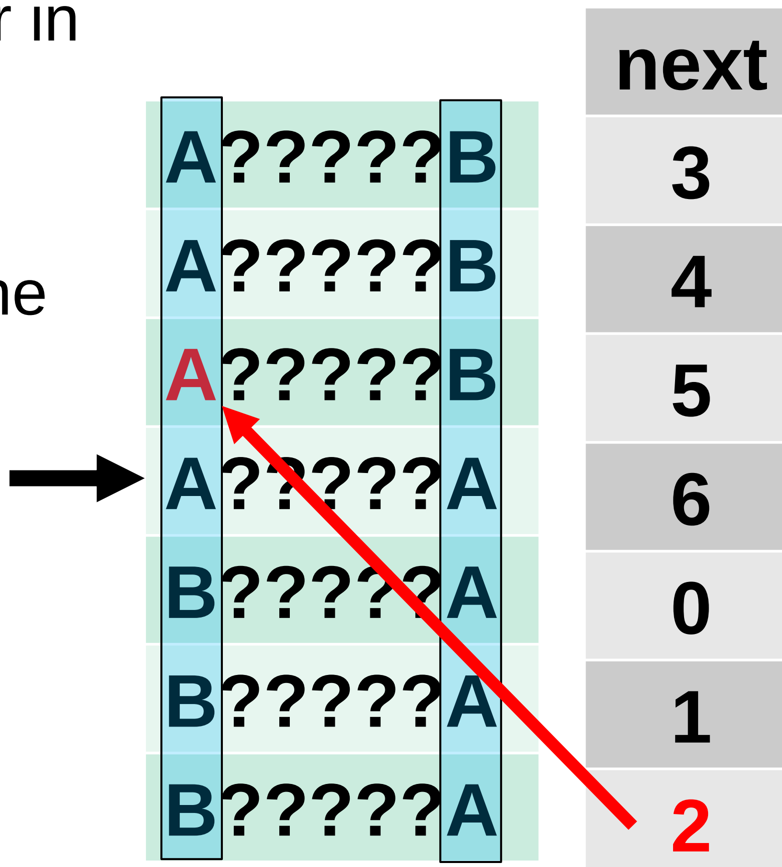
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[3]

AB?????



Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[6]

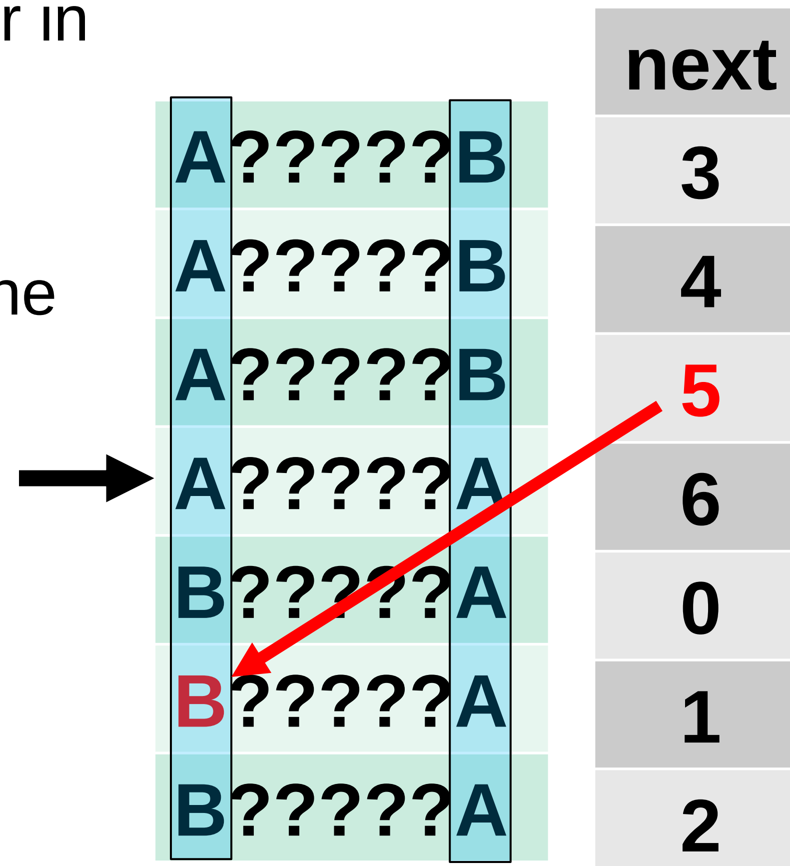


ABA????

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[2]

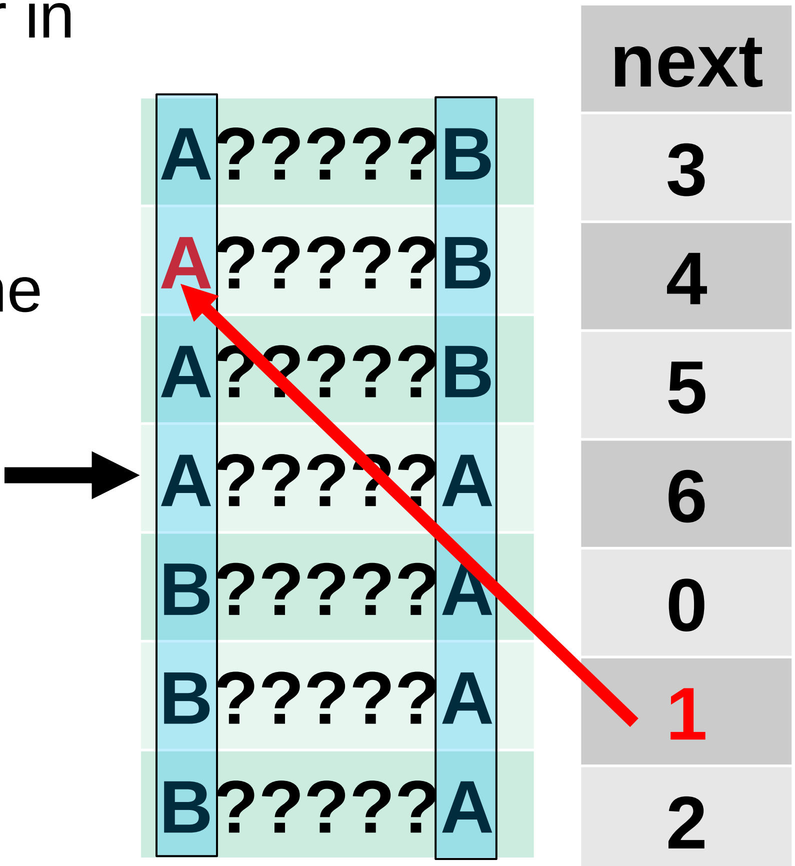
ABAB???



Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[5]

ABABA??



Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[5]

ABABABA

	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Downsides of Burrows-Wheeler Algorithm

- process **blocks** of input file
  - Compared to LZW, which processes the input **one character at time**
- The **larger** the block size, the **better** the compression
  - But the **longer** the sorting time

Repetitive Minimum Problem

- Input:
  - a (large) dynamic set of data items
- Output:
  - repeatedly find a minimum item
- You are implementing an algorithm that **repetitively** solve this problem
  - examples of such an algorithm?
    - Selection sort and Huffman tree construction
- What we cover today applies to the repetitive maximum problem as well

Let’s create an ADT!

- The Priority Queue ADT

- Let’s generalize min and max to highest **priority**
- Primary operations of the PQ:
  - Insert
  - Find item with highest priority
    - e.g., findMin() or findMax()
  - Remove an item with highest priority
    - e.g., removeMin() or removeMax()
- We mentioned priority queues in building Huffman tries
- How do we implement these operations?
  - Simplest approach: arrays

Unsorted array PQ

- Insert:
  - Add new item to the end of the array
  - $\Theta(1)$
- Find:
  - Search for the highest priority item (e.g., min or max)
  - $\Theta(n)$
- Remove:
  - Search for the highest priority item and delete
  - $\Theta(n)$

Sorted array PQ

- Insert:
  - Add new item in appropriate sorted order
  - $\Theta(n)$
- Find:
  - Return the item at the end of the array
  - $\Theta(1)$
- Remove:
  - Return and delete the item at the end of the array
  - $\Theta(1)$

So what other options do we have?

- What about a balanced binary search tree?
  - Insert
    - $\Theta(\lg n)$
  - Find
    - $\Theta(\lg n)$
  - Remove
    - $\Theta(\lg n)$
- OK, all operations are $\Theta(\lg n)$
  - No constant time operations

Which implementation should we choose?

- Depends on the application
- We can compare the *amortized runtime* of each implementation
- Given a set of operations performed by the application:

$$\text{Amortized runtime} = \frac{\text{Total runtime of a sequence of operations}}{\#\text{operations}}$$

Example: Huffman Trie Construction

- K-1 iterations
  - K is the # unique characters in the file to be compressed
- Each iteration:
  - 2 removeMin calls
  - 1 insert call
- Unsorted Array: Total time Huffman Trie Construction $=(K-1)*[2 * K + 1 * 1] = O(K^2)$
- Sorted Array: Total time Huffman Trie Construction $=(K-1)*[2 * 1 + 1 * K] = O(K^2)$
- Balanced BST: Total time Huffman Trie Construction $=(K-1)*[2 * \log K + 1 * \log K] = O(K \log K)$

Repetitive Highest Priority Problem

- Input:
  - a (large) dynamic set of data items
    - each item has a priority
    - e.g., highest priority is minimum item
    - e.g., highest priority is maximum item
  - a *stream* of zero or more of each of the following operations
    - Find a highest priority item in the set
    - Insert an item to the set
    - Remove a highest priority item from the set
- Examples
  - Selection sort
    - Repeatedly, remove a minimum item from the array and insert it in its correct position in the sorted array
  - Huffman trie construction
    - Each iteration: remove a minimum tree from the forest (**twice**) and insert a new tree

Let's create an ADT!

- The ADT Priority Queue (PQ)

- Primary operations of the PQ:
  - Insert
  - Find item with highest priority
    - e.g., findMin() or findMax()
  - Remove an item with highest priority
    - e.g., removeMin() or removeMax()

What are possible implementations of the PQ ADT?

	findMin	removeMin	insert
Unsorted Array	O(n)	O(n)	O(1)
Sorted Array	O(1)	O(1)	O(n)
Red-Black BST	O(log n)	O(log n)	O(log n)

Is a BST overkill to implement ADT PQ?

- Balanced BST (e.g., RB-BST) provides *log n* runntime time for all operations
- Our find and remove operations only need the highest priority item, not to find/remove *any* item
  - Can we take advantage of this to improve our runtime?
    - Yes!

The heap

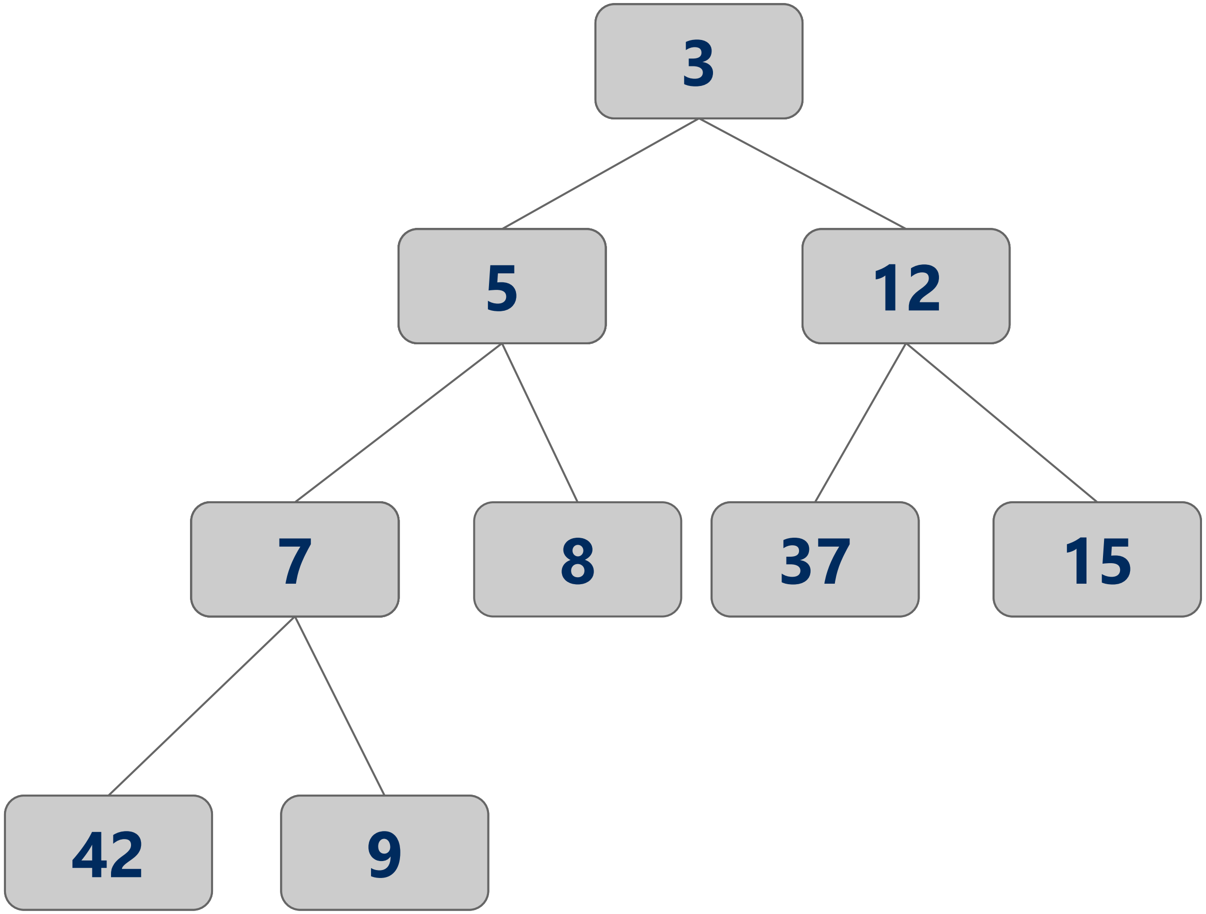
- A heap is **complete** binary tree such that for each node T in the tree:
  - T.item is of a higher priority than T.right_child.item
  - T.item is of a higher priority than T.left_child.item

- It does not matter how T.left_child.item relates to T.right_child.item
  - This is a relaxation of the approach needed by a BST

The *heap property*

Min Heap Example

- In a Min Heap, a highest priority item is a minimum item



Heap PQ runtimes

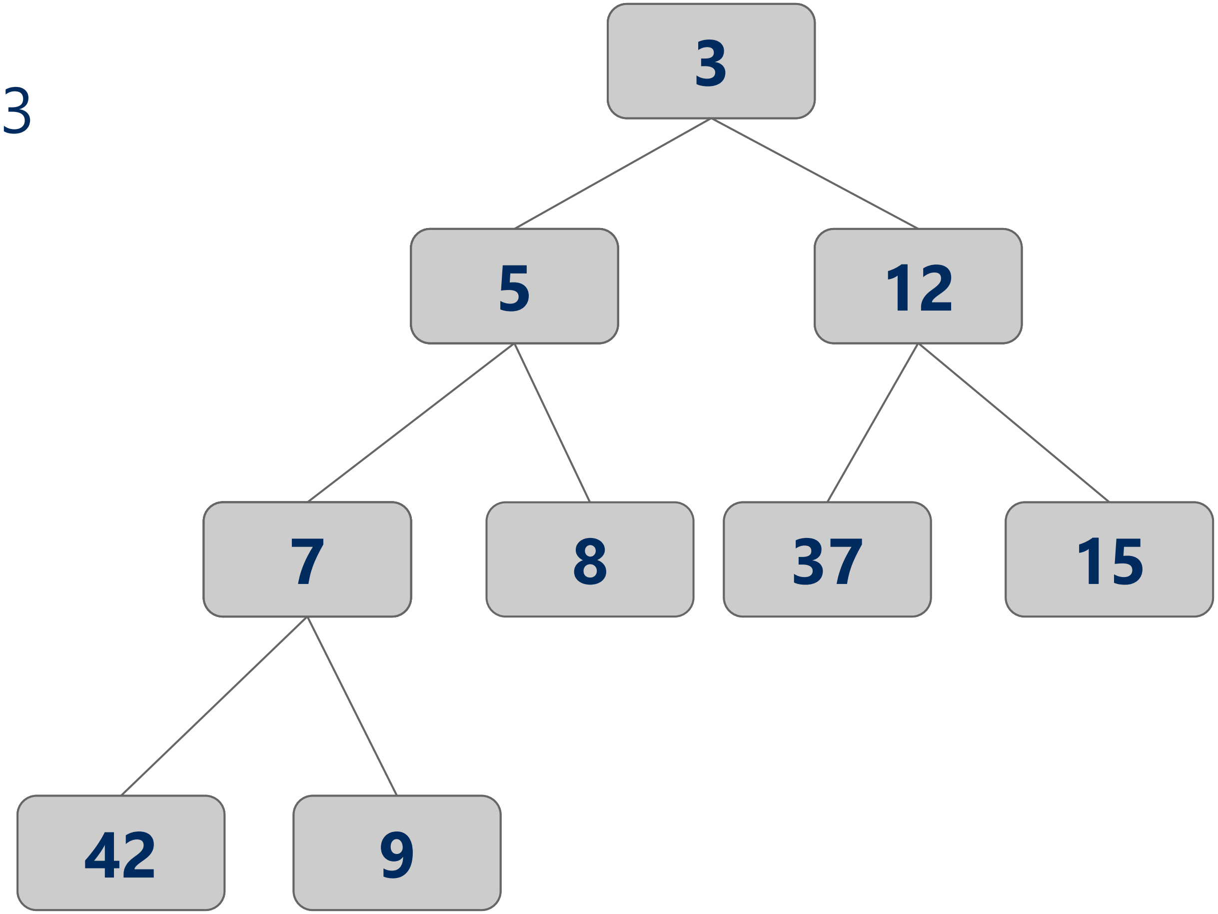
- Find is easy
  - Simply the root of the tree
    - $\Theta(1)$
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained

Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3

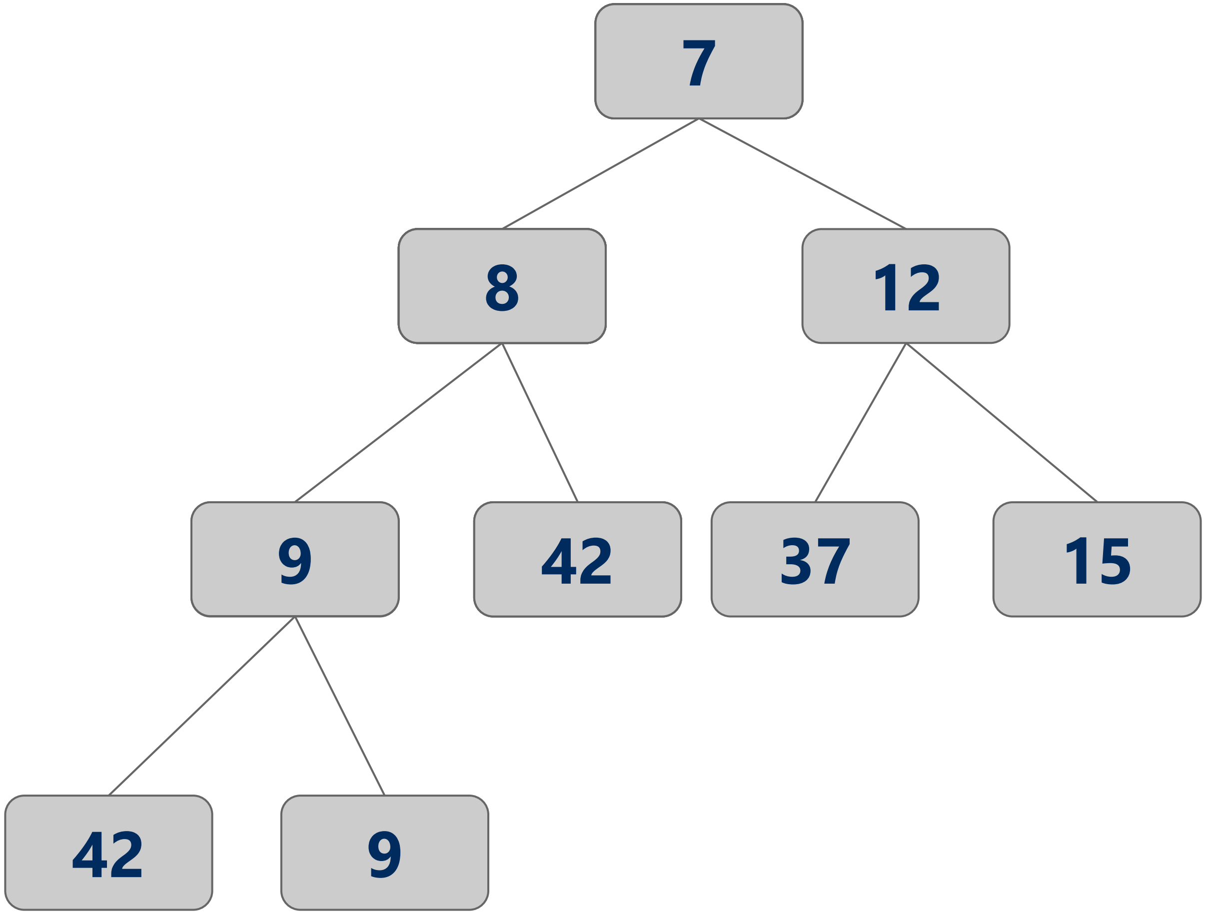


Heap remove

- Tricky to delete root...
  - So let's simply overwrite the root with the item from the last leaf and delete the last leaf
    - But then the root is violating the heap property...
      - So we push the root down the tree until it is supporting the heap property

Min heap removal

NO!



Heap runtimes

- Find
  - $\Theta(1)$
- Insert and remove
  - Height of a complete binary tree is lg n
  - At most, upheap and downheap operations traverse the height of the tree
  - Hence, insert and remove are $\Theta(\lg n)$

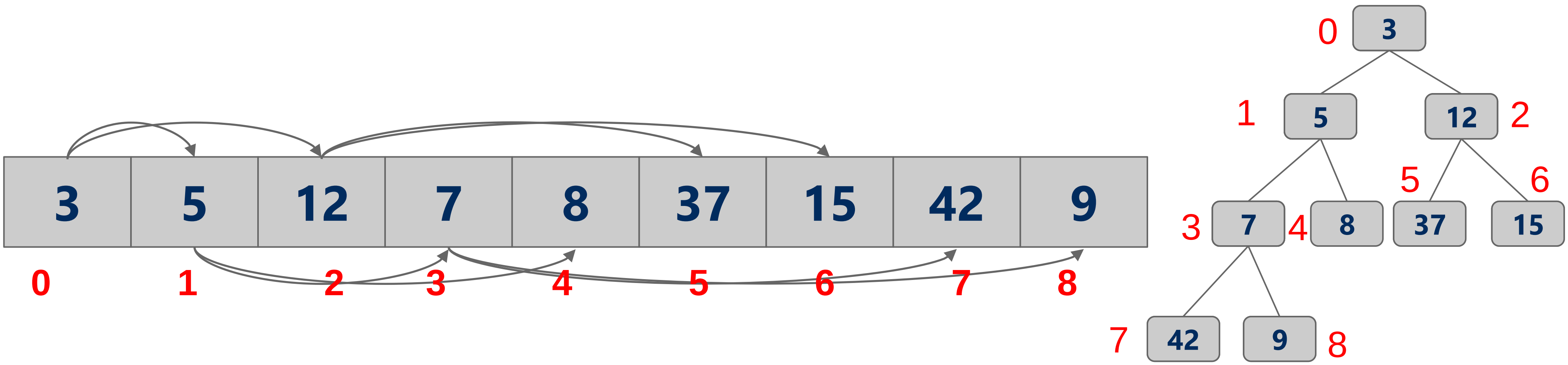
Heap implementation

- Simply implement tree nodes like for BST
    - This requires overhead for dynamic node allocation
    - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree…
    - We can easily represent a complete binary tree using an array

Storing a heap in an array

- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index i
  - parent(i) = $\lfloor (i - 1) / 2 \rfloor$
  - left_child(i) = 2i + 1
  - right_child(i) = 2i + 2

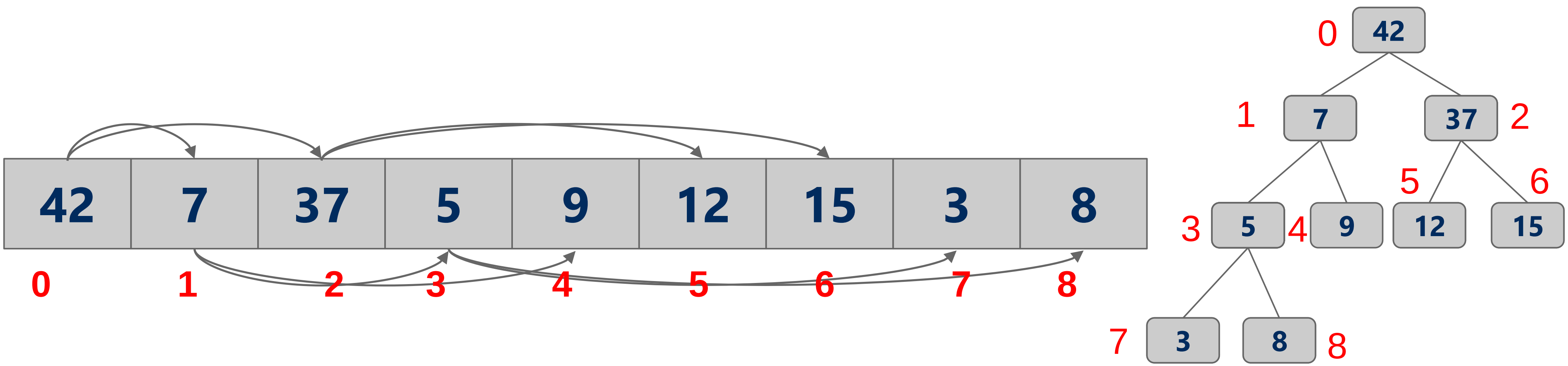
For arrays indexed from 0



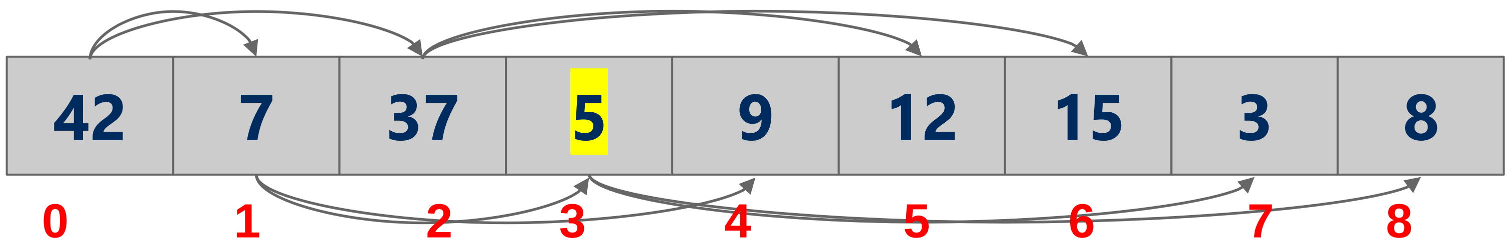
Can we turn any array into a heap?

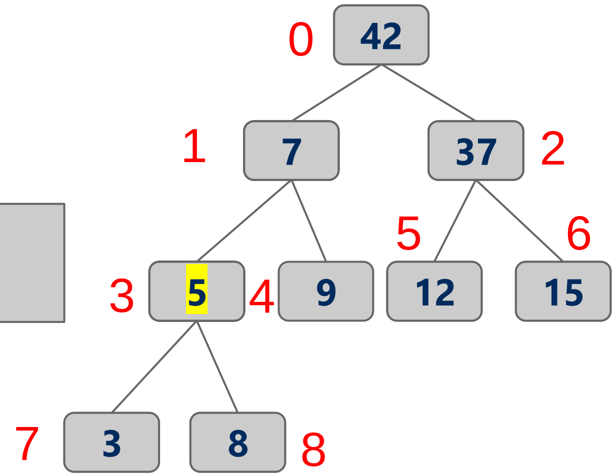
- Yes!
- Any array can be thought of as a complete tree!
- We can change it into a heap using the following algorithm
- Scan through the array **right to** left starting from the rightmost non-leaf
  - the largest index *i* such that left_child(i) is a valid index (i.e., < n)
  - $2i+1 < n \rightarrow i < (n-1)/2$
  - push the node down the tree until it is supporting the heap property
- This is called the **Heapify** operation

Heapify Example: Building a Min Heap

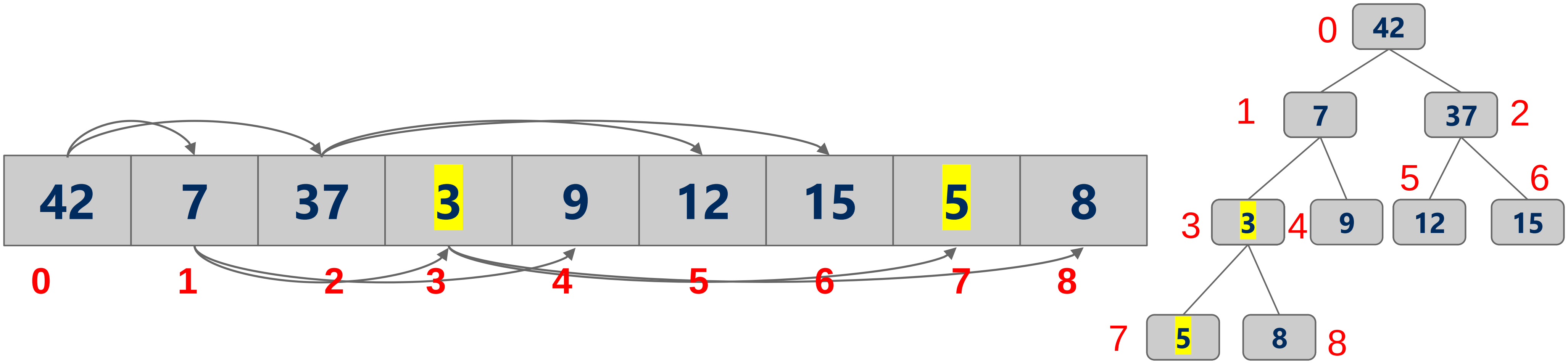


Heapify Example: Building a Min Heap

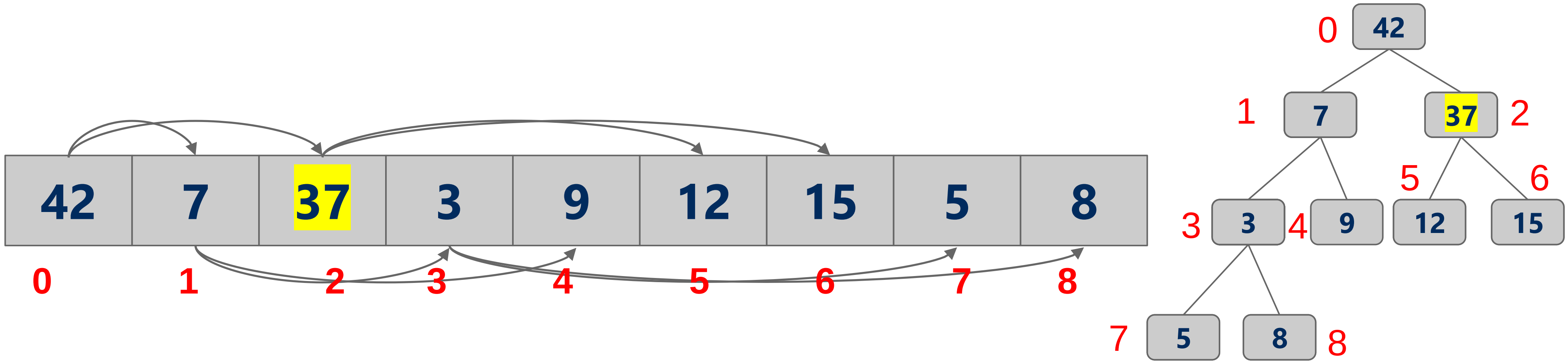




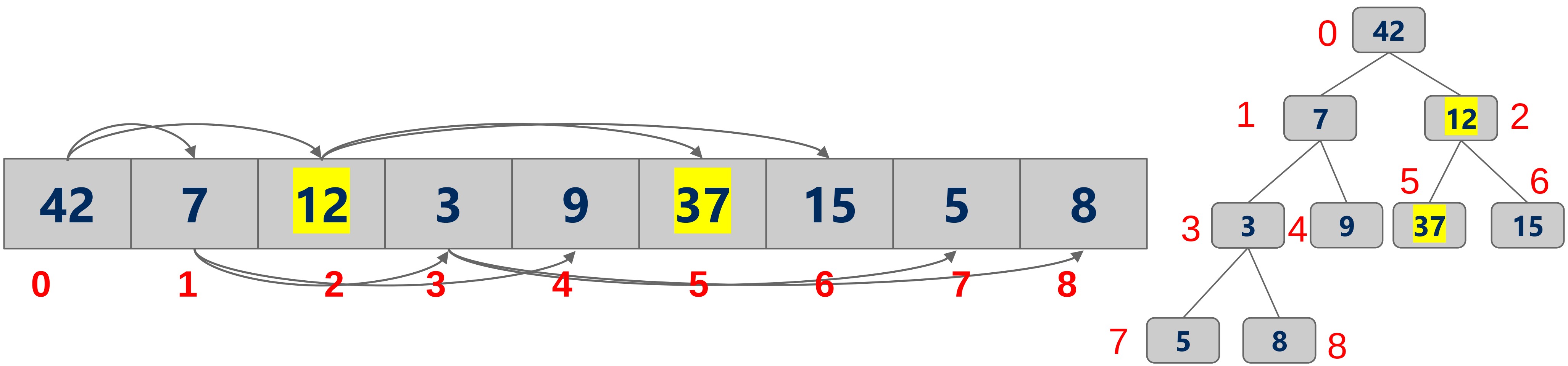
Heapify Example: Building a Min Heap



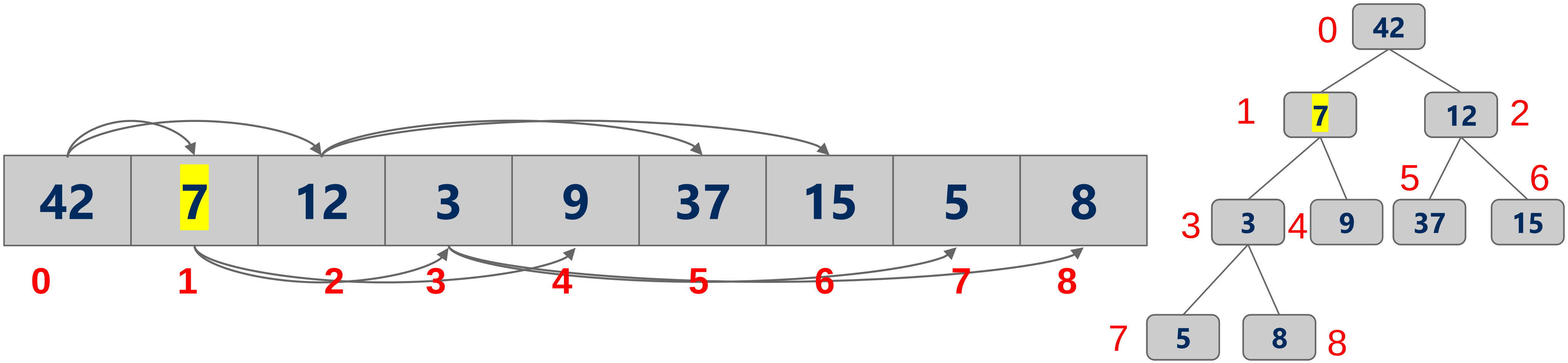
Heapify Example: Building a Min Heap



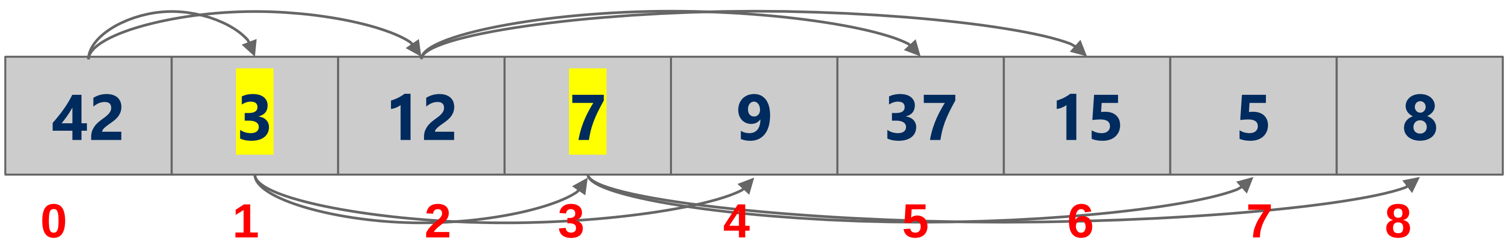
Heapify Example: Building a Min Heap

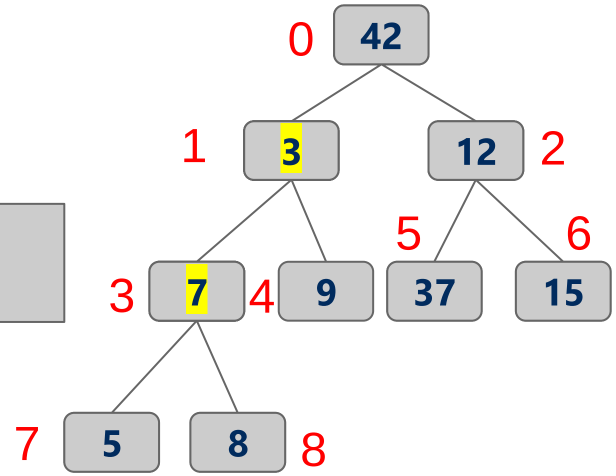


Heapify Example: Building a Min Heap

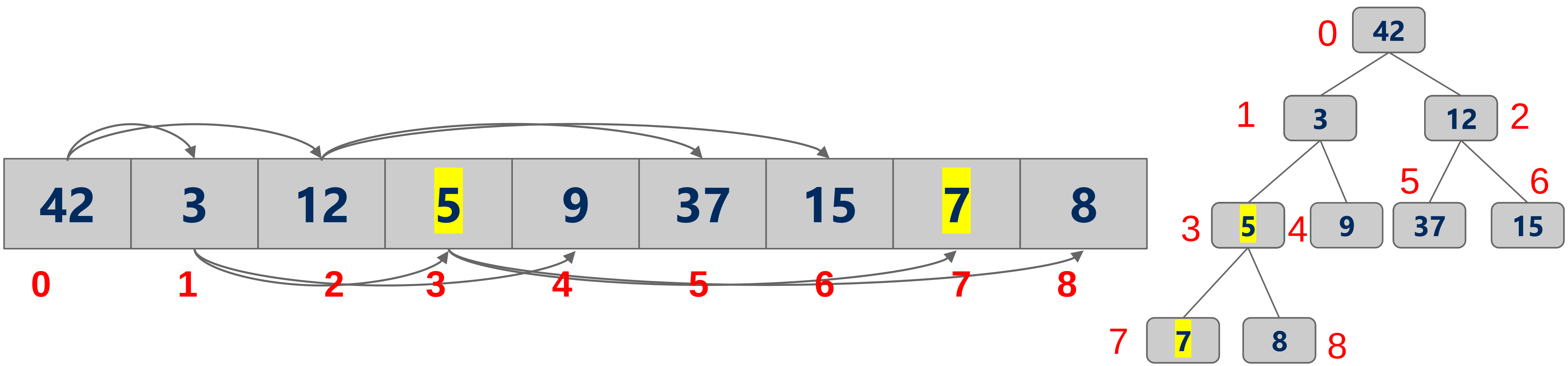


Heapify Example: Building a Min Heap

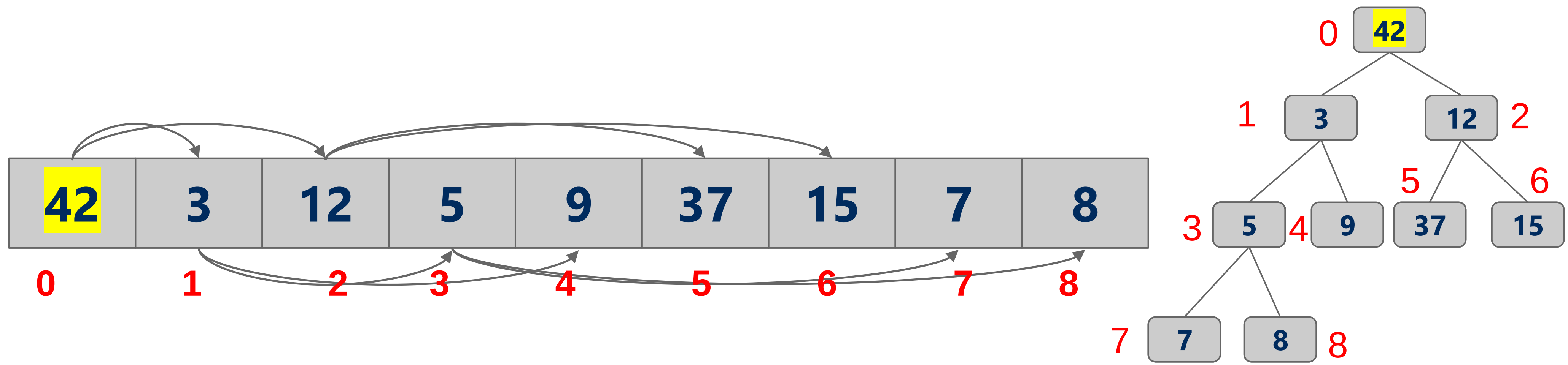




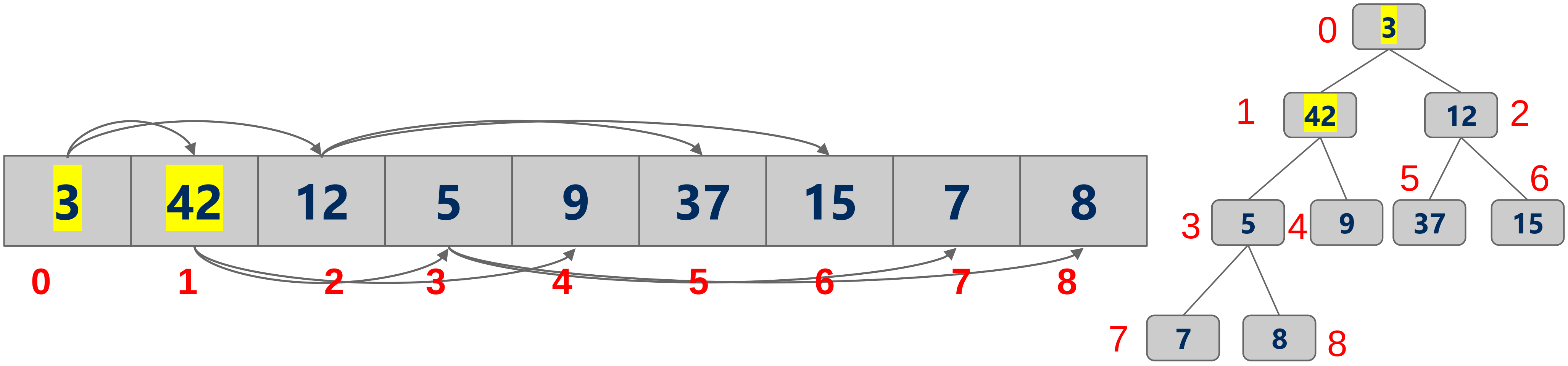
Heapify Example: Building a Min Heap



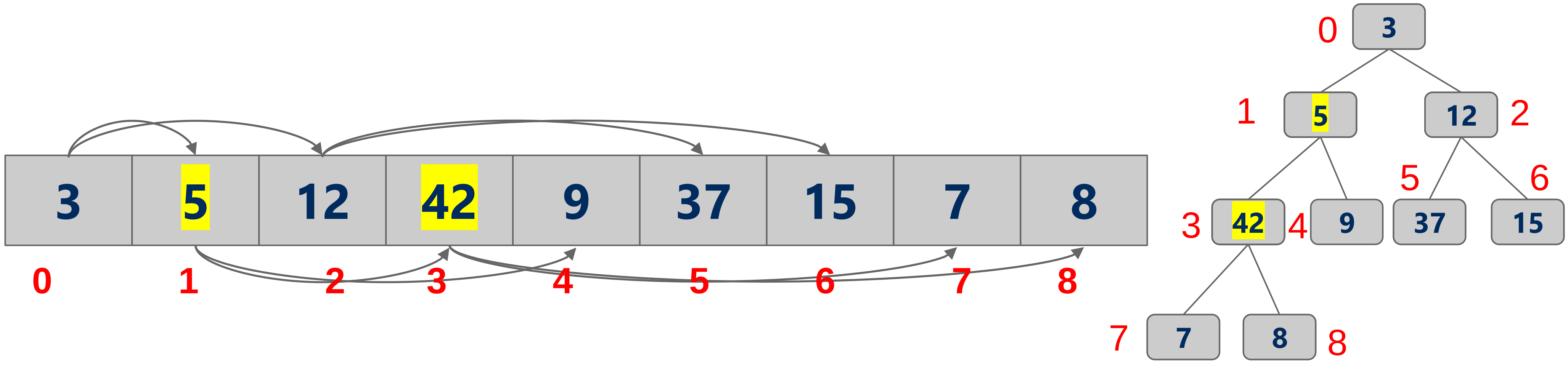
Heapify Example: Building a Min Heap



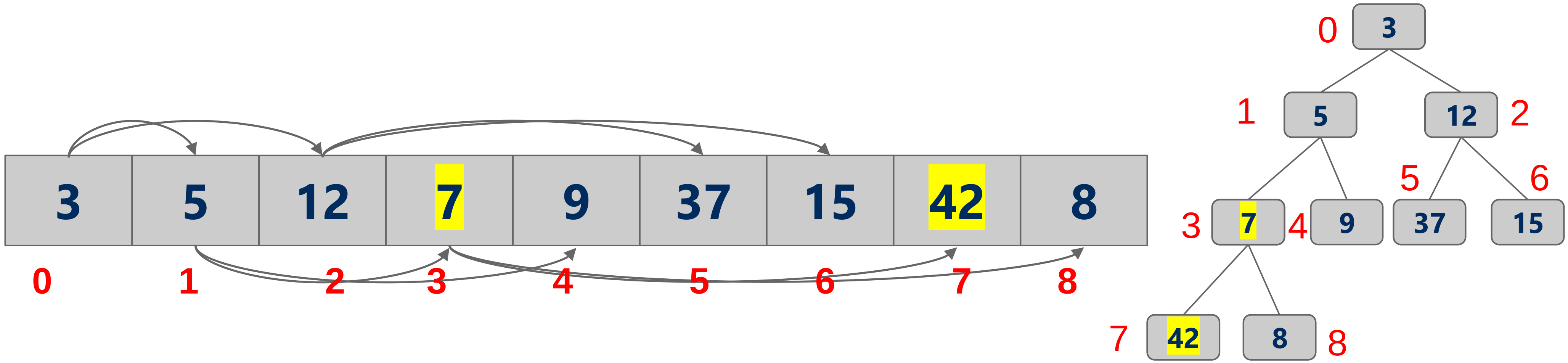
Heapify Example: Building a Min Heap



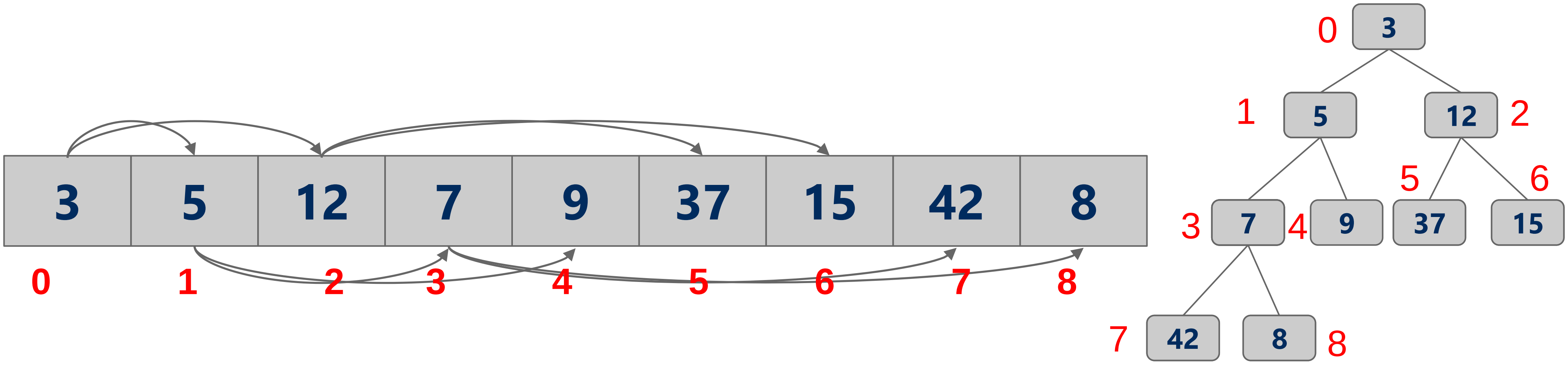
Heapify Example: Building a Min Heap



Heapify Example: Building a Min Heap



Heapify Example: Building a Min Heap

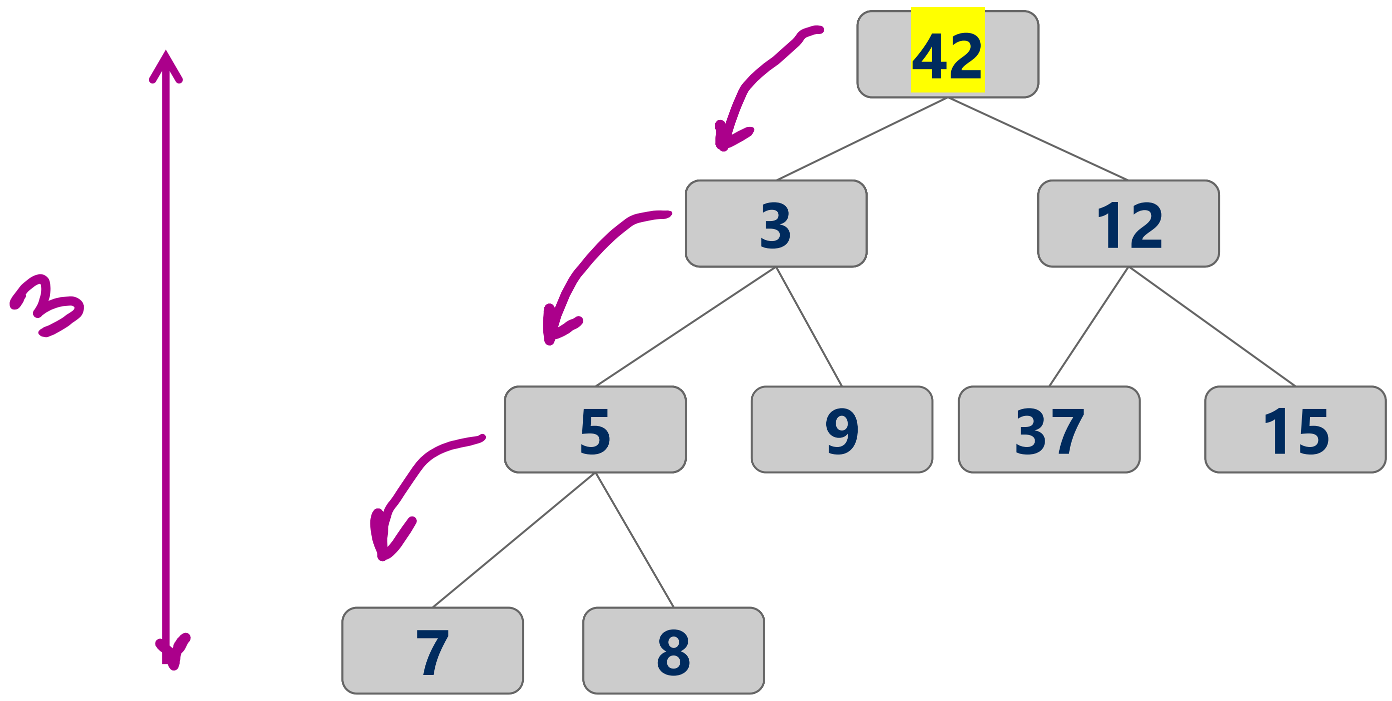


Heapify Running time

- Upper bound analysis:
    - We make about n/2 downheap operations
        - log n each
    - So, O(n log n)

Heapify Running time

- A tighter analysis
  - for each node that we start from, we make at most *height[node]* swaps



Heapify Running time: A tighter analysis

- $Runtime = \sum_{i=1}^{n} height[n]$
- $= \sum_{i=0}^{\log n} number\ of\ nodes\ with\ height\ i$
- Assume a full tree
  - A node with height *i* has 2^i nodes in its subtree including itself
  - Assume *k* nodes with height *i*:
  - they will have $k2^i$ nodes in their subtrees
  - $k2^i <= n \rightarrow k <= n/2^i$
- So, at most $n/2^i$ nodes exist with height *I*
- $\sum_{i=o}^{\log n} \frac{n}{2^i} = n + \frac{n}{2} + \frac{n}{4} + \ \dots$
- $= \ \theta(largest\ term) = \ \theta(n)$

Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- "Remove" the root
  - Don’t actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat

Heap sort analysis

- Runtime:
  - Worst case:
    - n log n
- In-place?
  - Yes
- Stable?
  - No

Storing Objects in PQ

- What if we want to **update** an Object in the heap?
    - What is the runtime to find an arbitrary item in a heap?
        - Θ(n)
        - Hence, updating an item in the heap is Θ(n)
    - Can we improve of this?
        - Back the PQ with something other than a heap?
        - Develop a clever workaround?

Indirection

- Maintain a second data structure that maps item IDs to each item's current position in the heap
- This creates an *indexable* PQ

Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

```
class CardPrice implements Comparable<CardPrice>{
    public String store;
    public double price;
    public CardPrice(String s, double p) { … }
    public int compareTo(CardPrice o) {
        if (price < o.price) { return -1; }
        else if (price > o.price) { return 1; }
        else { return 0; }
    }
}
```

Indirection example

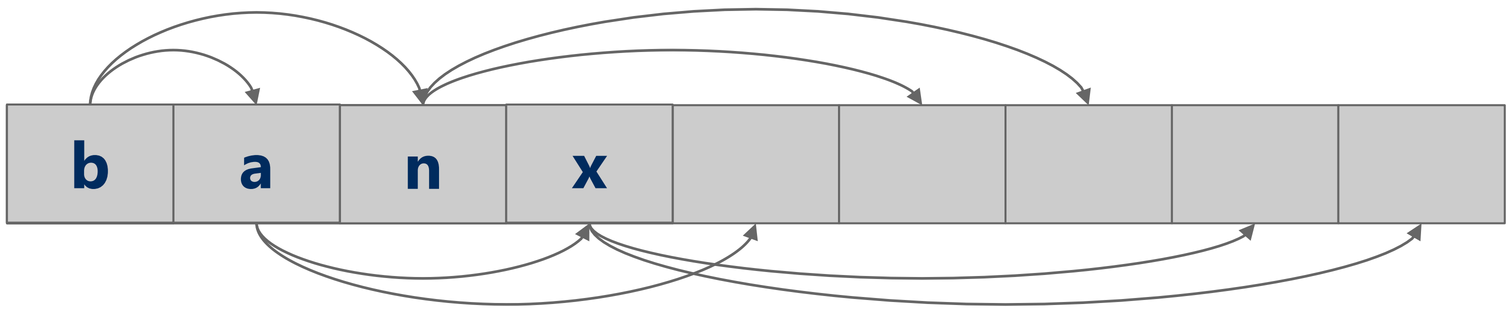
- n = new CardPrice("NE", 333.98);
- a = new CardPrice("AMZN", 339.99);
- x = new CardPrice("NCIX", 338.00);
- b = new CardPrice("BB", 349.99);

- Update price for NE: 340.00

- Update price for NCIX: 345.00

- Update price for BB: 200.00

Indirection
"NE":2
"AMZN":1
"NCIX":3
"BB":0



Indexable PQ Discussion

- How are our runtimes affected?
- space utilization?
- how should we implement the indirection?
- what are the tradeoffs?

A new problem!!

- **Input**: A file containing LinkedIn (LI) accounts and their connections
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

Problem of the Day

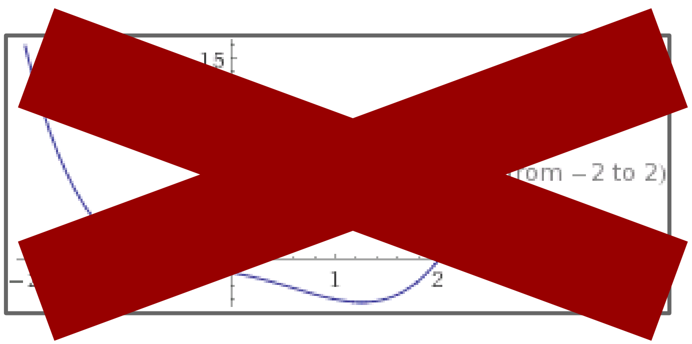
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - e.g., 1st connection?, 2nd connection?, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - For each connected component, are there certain accounts that if removed, the remaining accounts become ***partitioned***?

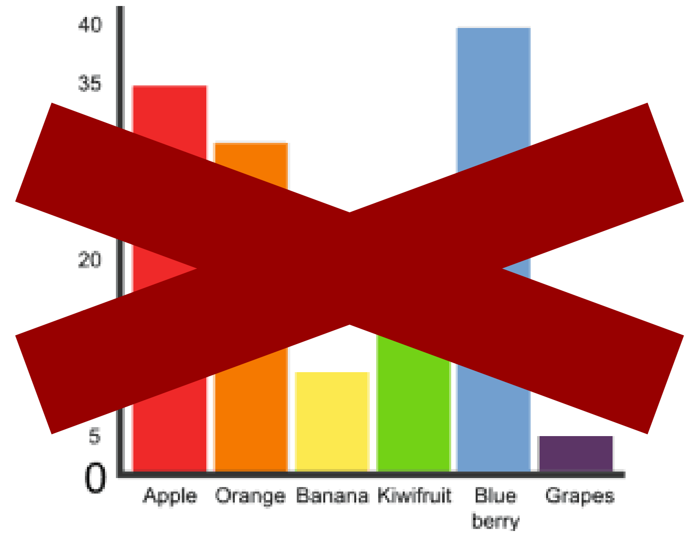
Which Data Type to use?

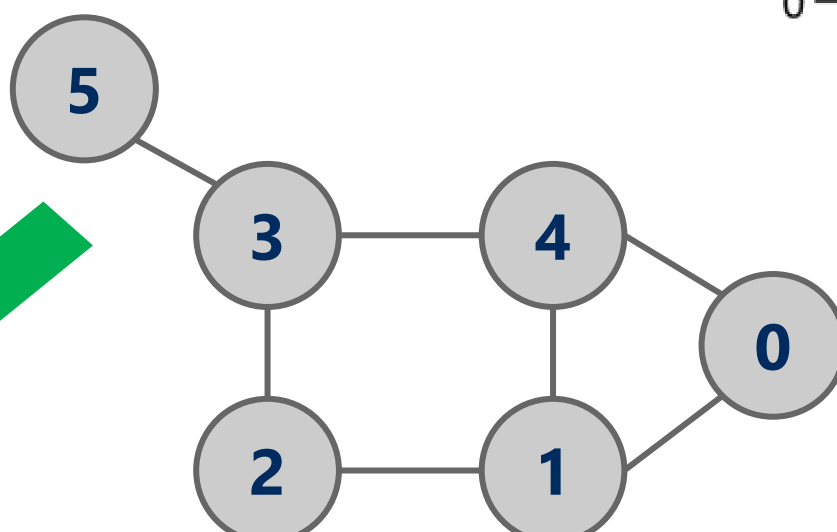
- Let's think first about how to organize the data that we have in memory
- Note that the operations are different from what we have been used to (search, sort, min, max, add, delete, …)

- Account1: Connection1, Connection2, …
- Account2: Connection1, Connection2, …
- …

Graphs!

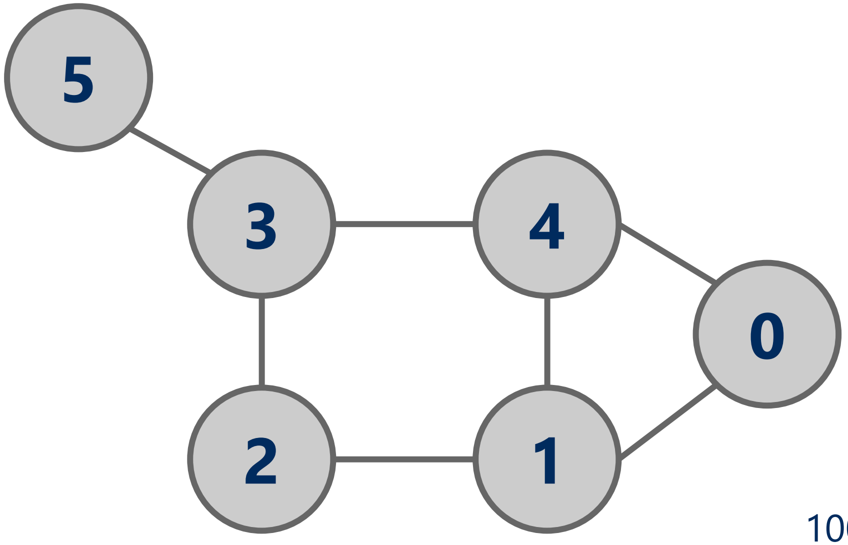



40
35
20
5
0
Apple
Orange
Banana
Kiwifruit
Blue berry
Grapes


5
3
4
0
2
1

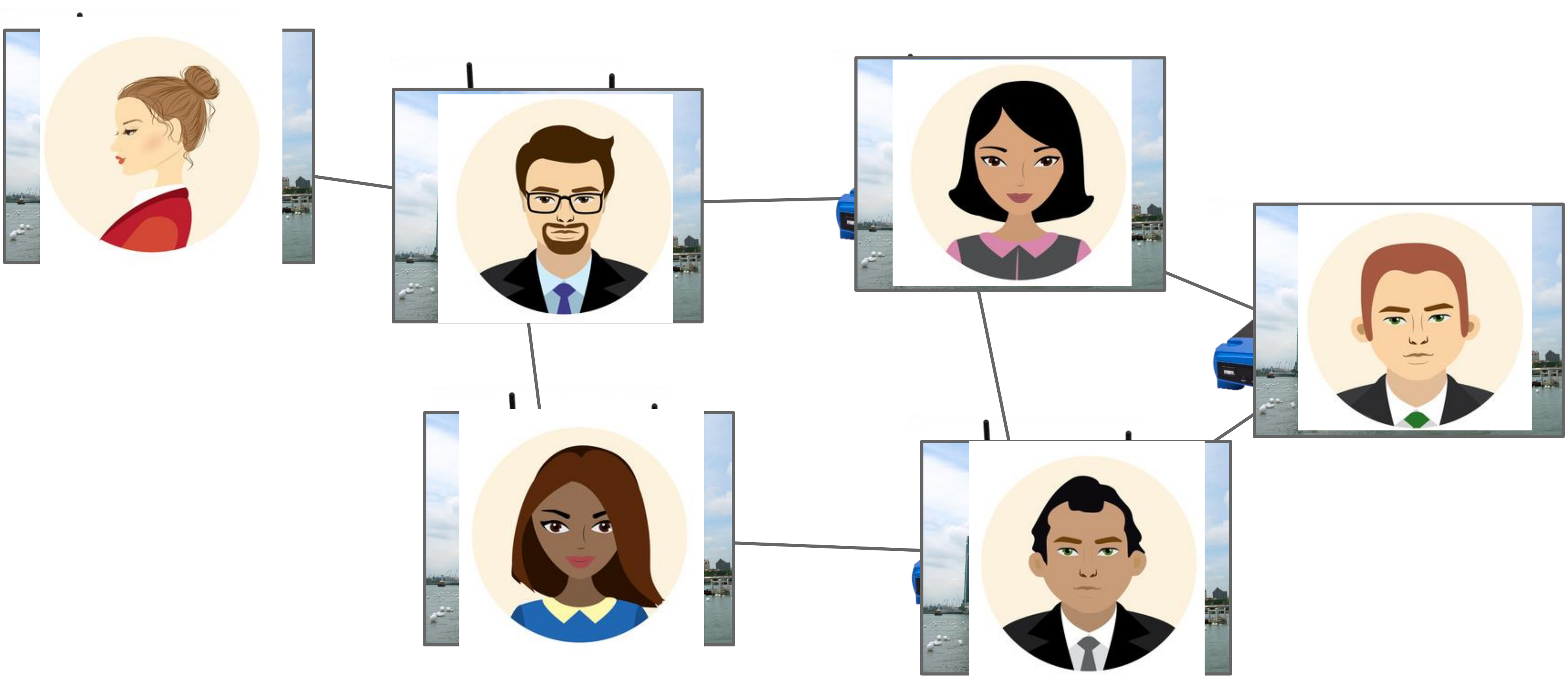
Graphs

- A graph G = (V, E)
  - where V is a set of vertices
  - E is a set of edges connecting vertex pairs
- Example:
  - V = {0, 1, 2, 3, 4, 5}
  - E = {(0, 1), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4), (3, 5)}



Why?

- Can be used to model many different scenarios



Some definitions

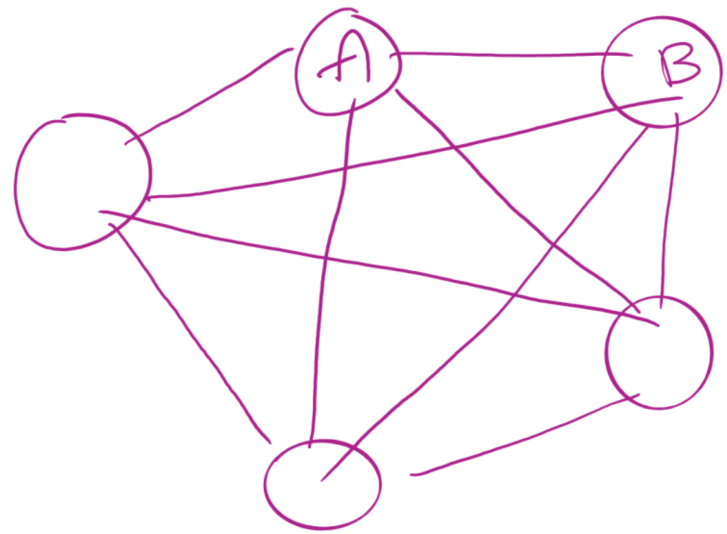
- Undirected graph
  - Edges are unordered pairs: (A, B) == (B, A)
- Directed graph
  - Edges are ordered pairs: (A, B) != (B, A)
- Adjacent vertices, or neighbors
  - Vertices connected by an edge

Graph sizes

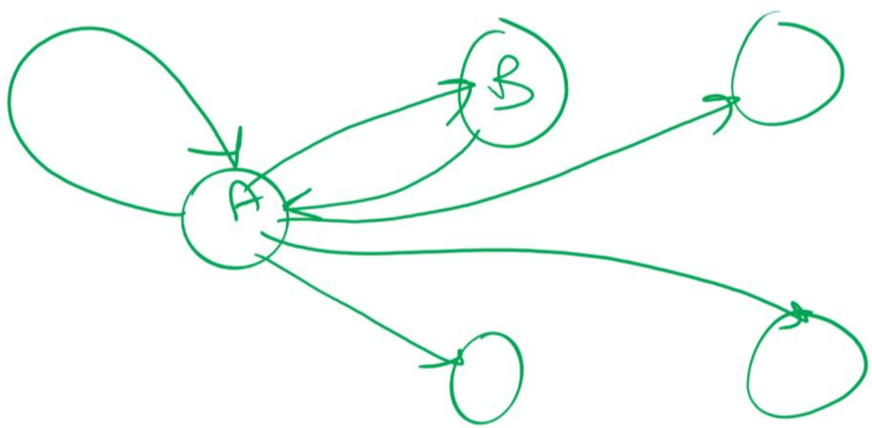
- Let v = |V|, and e = |E|
- Given v, what are the minimum/maximum sizes of e?
  - Minimum value of e?
    - Definition doesn't necessitate that there are any edges…
    - So, 0
  - Maximum of e?
    - Depends…
      - Are self edges allowed?
      - Directed graph or undirected graph?
    - In this class, we'll assume directed graphs have self edges while undirected graphs do not

Maximum value of e (MAX)

- Undirected graph
  - no self edges
  - v*(v-1)?
  - But, A->B is the same edge as B-> A
  - Are we counting each twice?
  - v*(v-1)/2

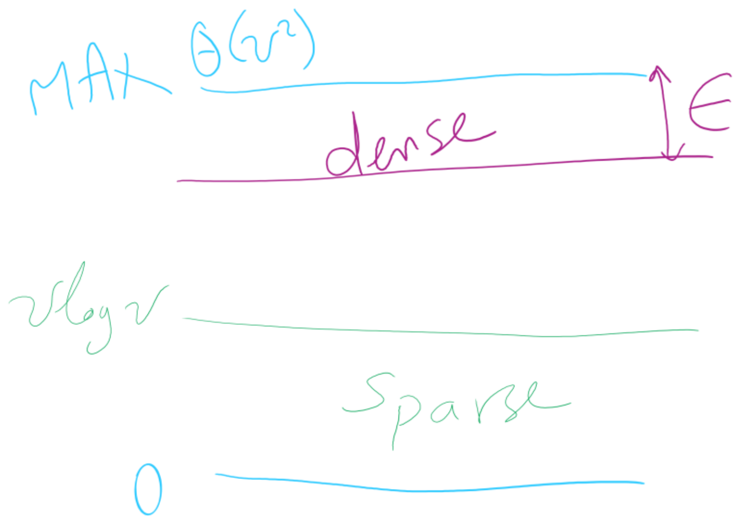


- Directed graph
  - self edges allowed
  - v*v?
  - A -> B is a different edge than B -> A
  - v^2

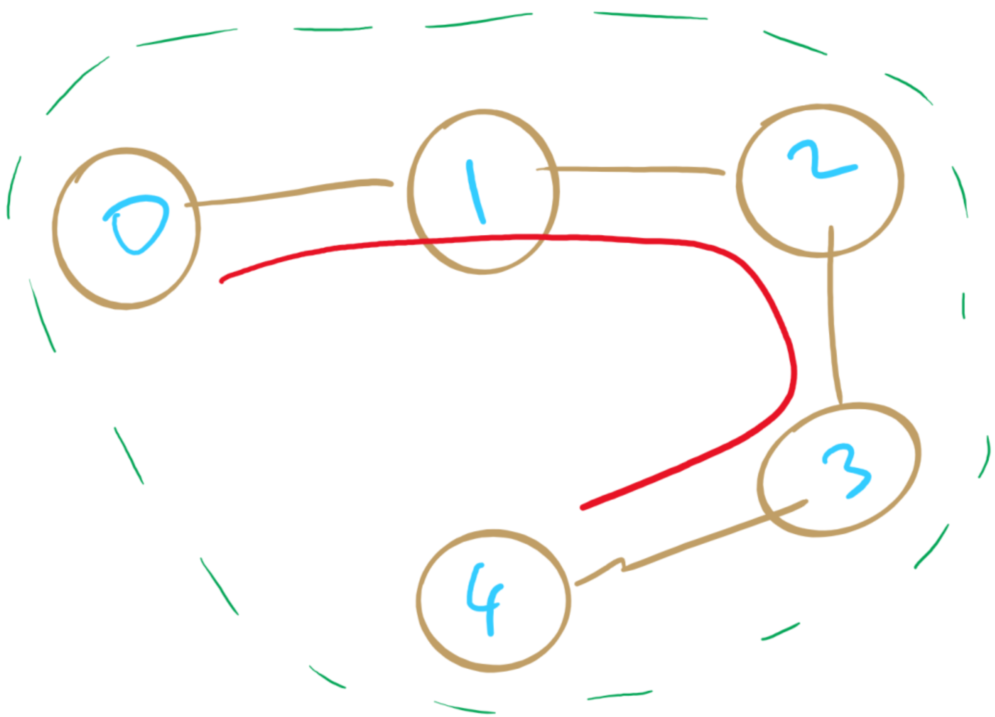


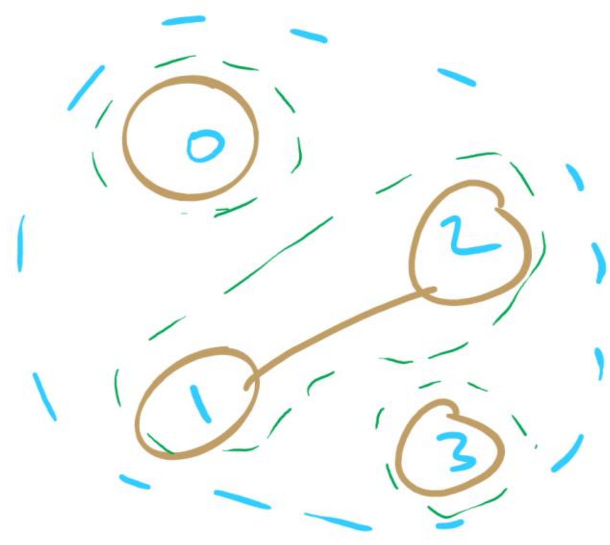
More definitions

- A graph is considered *sparse* if:
    - e <= v lg v
- A graph is considered *dense* as it approaches the maximum number of edges
    - I.e., e == MAX - ε
- A *complete* graph has the maximum number of edges
- Have we seen “sparse” and dense before?



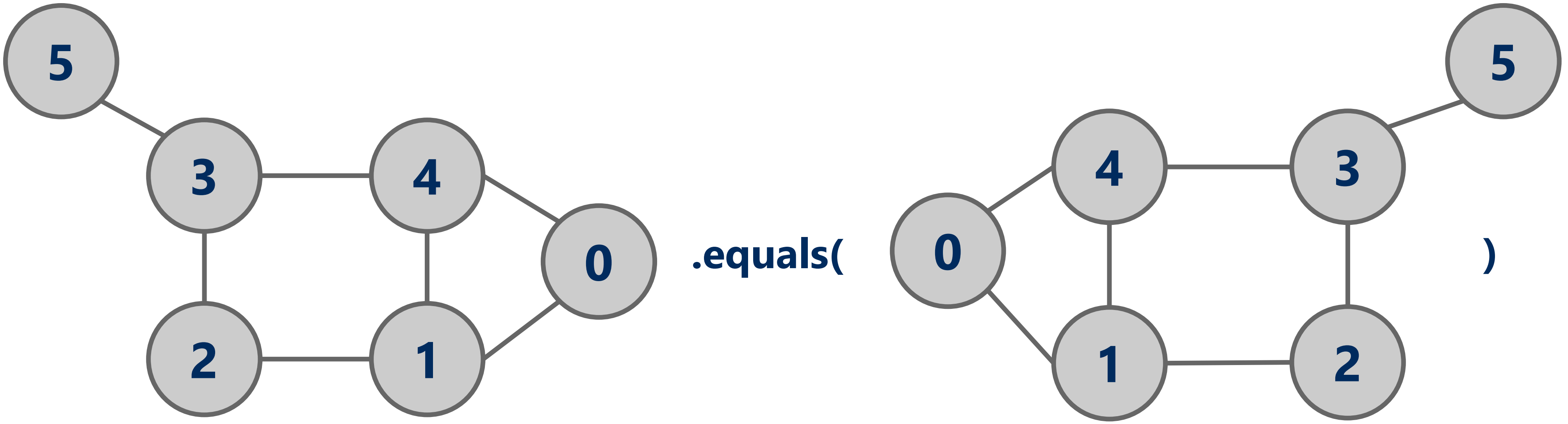
Sparse graphs





Question:

- Is



Representing graphs

- Trivially, graphs can be represented as:
  - List of vertices
  - List of edges
- Performance?
  - Assume we're going to be analyzing static graphs
    - I.e., no insert and remove
  - So what operations should we consider?

Graph operations

- Static graphs
  - check if two vertices are neighbors
  - find the list of neighbors of a given vertex
    - for directed graphs, in-neighbors and out-neighbors
- Dynamic graphs
  - add/remove edges
  - Not our focus in this class

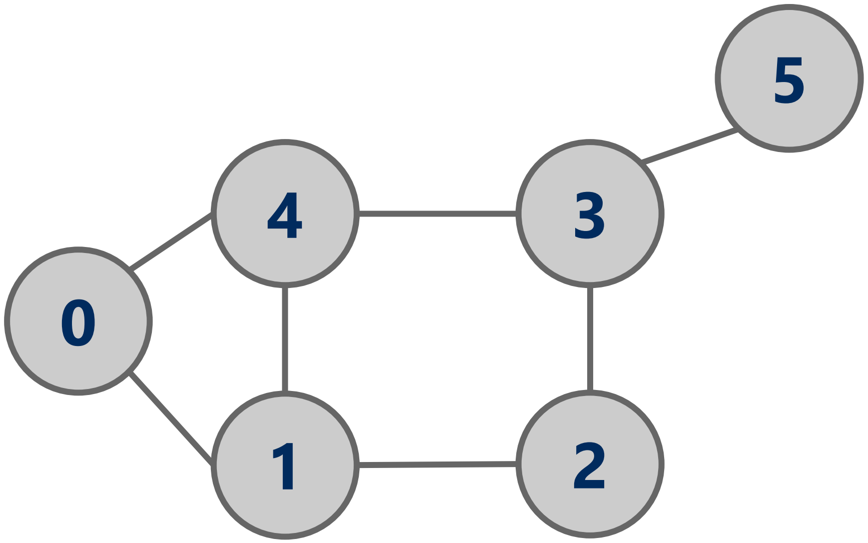
Representing graphs

- Trivially, graphs can be represented as:
  - List of vertices
  - List of edges
- Performance?
  - Check if two vertices are neighbors
    - $O(e)$
  - Find the list of neighbors of a given vertex
    - $O(e)$
- Space?
  - $\Theta(v + e)$ memory

Using an adjacency matrix

- Rows/columns are vertex labels
  - M[i][j] = 1 if $(i, j) \in E$
  - M[i][j] = 0 if $(i, j) \notin E$

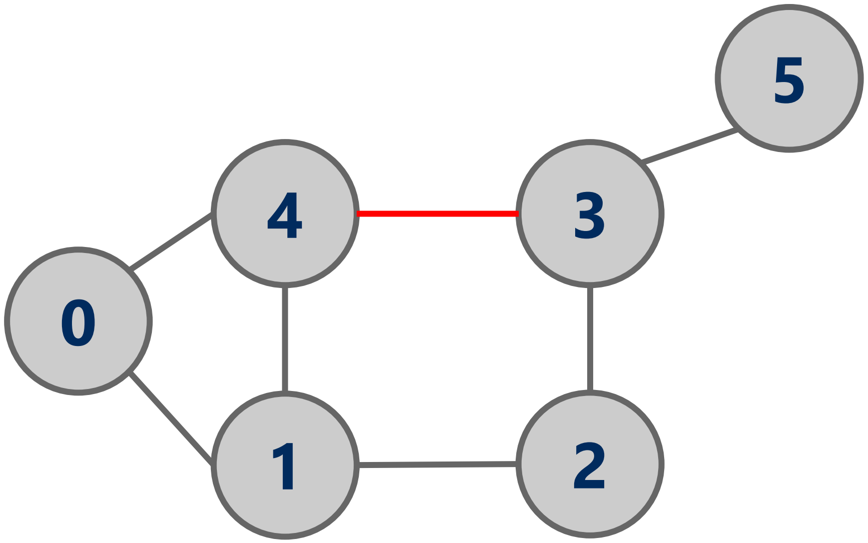
	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0



Using an adjacency matrix

- Rows/columns are vertex labels
  - $M[i][j] = 1$ if $(i, j) \in E$
  - $M[i][j] = 0$ if $(i, j) \notin E$

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0



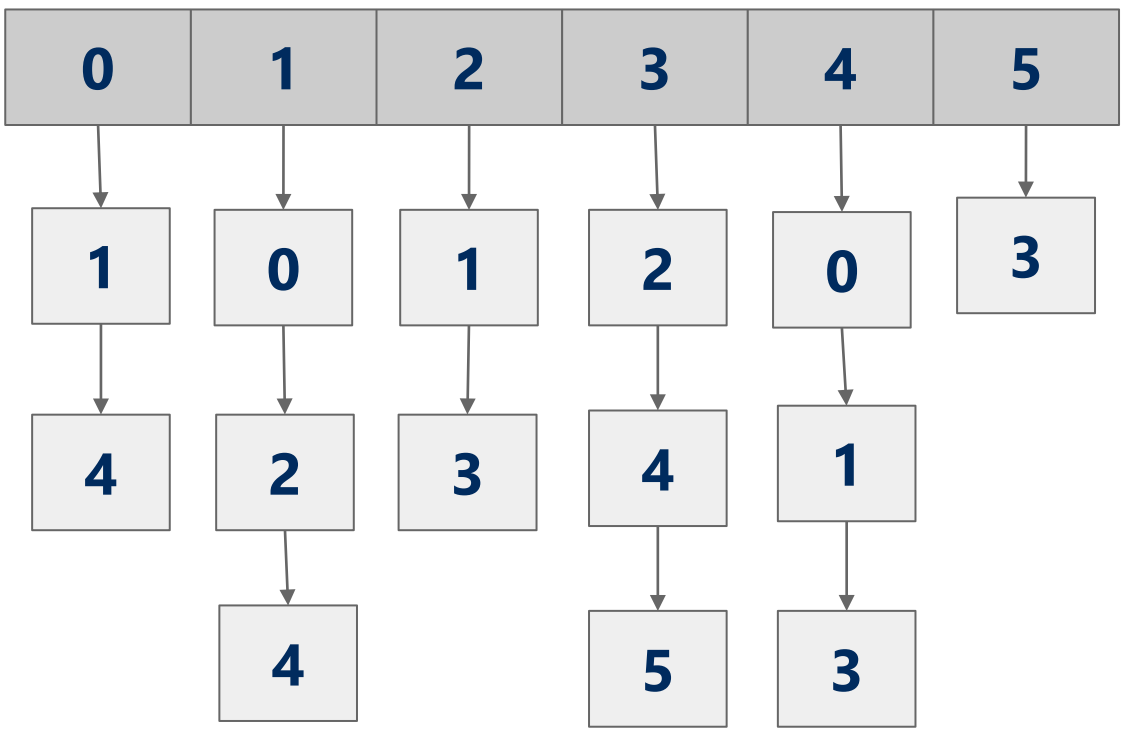
Adjacency matrix analysis

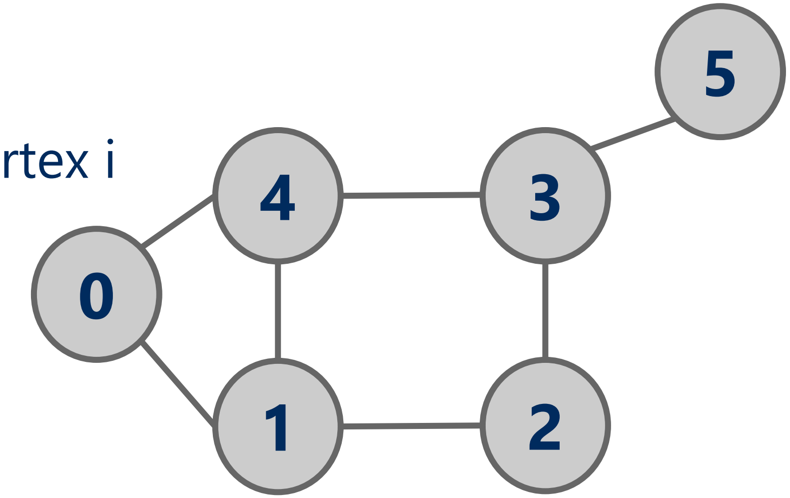
- Runtime?
  - Check if two vertices are neighbors
    - $\Theta(1)$
  - Find the list of neighbors of a vertex
    - $O(v)$
  - $O(v^2)$ time to initialize
- Space?
  - $O(v^2)$

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0

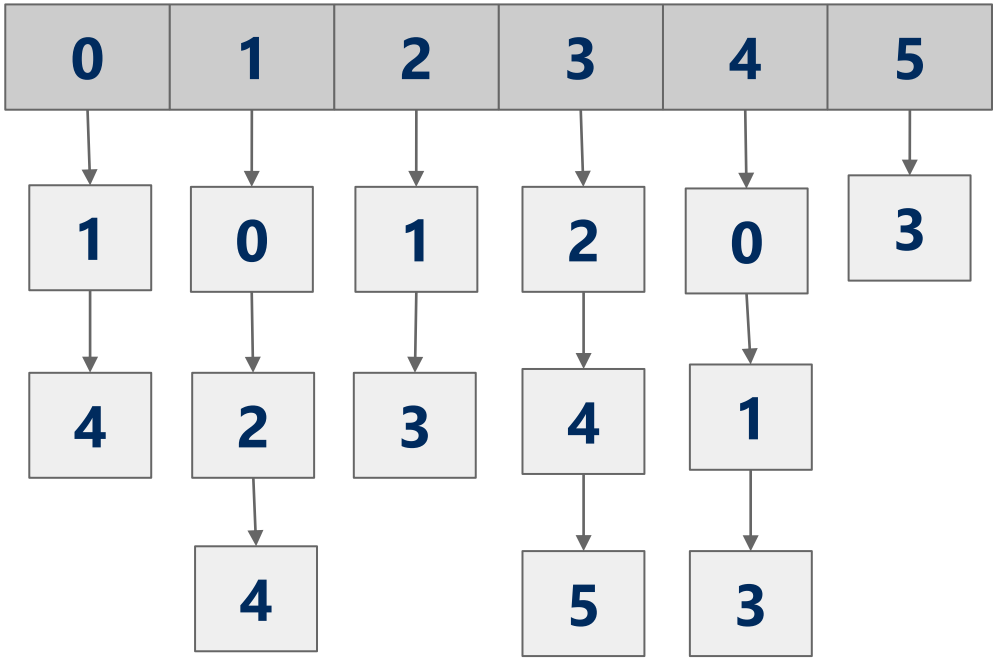
Adjacency lists

- Array of neighbor lists
  - A[i] contains a list of the neighbors of vertex i





Adjacency list analysis



- Runtime?
  - Check if two vertices are neighbors
  - Find the list of neighbors of a vertex
    - Θ(d)
    - d is the degree of a vertex (# of neighbors)
    - O(v)
- Space?
  - Θ(v + e) memory
  - overhead of node use
  - Could be much less than v^2

Comparison

- **Where would we want to use adjacency lists vs adjacency matrices?**
- Dense graphs?
- Sparse graphs?
- **What about the list of vertices/list of edges approach?**